Application of Neutral Networks to Regression and Classification Problems
FYS-STK3155 - Project 2

Alexander Umansky
Norwegian University of Science and Technology, University of Oslo
(Dated: October 31, 2025)

In this study, we developed a Neural Network with customizable Feed Forward architecture, without convolutional or recurrent filtering stages. The training algorithm uses Gradient Descend compatible with adaptive learning rate, regularization and batch sampling. The Neural Network was applied to fit non-trivial shapes in two dimensions, as well as evaluating binary and multi-class datasets. Several architecture and optimization attributes were tested to achieve best shape tracing or classification accuracy. ADAM with learning rate (...) was preferred for its handling of complex phase space. Regression with Neural Networks surpassed the traditional method for one-dimensional datasets of ($N > \dots$) even when stronger noise ($\sigma = \dots$) was present. In higher dimension overfitting was reduced using regularization (...) and redefining the architecture (...). In classification without convolutional filtering, we relied on regularization, and manual feature selection using correlation matrices. Good accuracy was achieved when using (...) layers and (adapted Softmax) activation functions.

I. INTRODUCTION

Artificial Neural Networks (ANNs) are inspired by biological information processing inside the human nervous system. Each neuron cell has a receiver and transmitter which forwards the signal once a certain activation potential is reached. Analogously, input data enters a network of interconnected nodes which can be regarded as the model itself. Transmission between connected nodes is controlled by activation functions and trigger parameters (weights and biases). Neural networks can[1] adapt to any functional input-output mapping, with right network architecture and optimized parameters, effectively shaping the geometry of the *responsive* model. This versatility is the reason ANNs are today applied in almost any type of problem, be it regression or classification. They are particularly powerful for identifying subtle and/or highly non-linear patterns in data, and the advantage over traditional methods only grows for higher dimensional data.

This study extends our previous work on performance of Gradient Descend optimization methods for polynomial regression[2]. There, a model of fixed complexity P was optimized to reproduce the non-trivial shape[2] of the Runge function $R(x) = \frac{1}{1+25x^2}, x \in (-1, 1)$ using a one-dimensional dataset perturbed by normally distributed noise $N(0, \sigma \leq 1)$. In the present work, we substitute the polynomial model with an arbitrary network architecture, and probe the Runge function in one and two dimensions $R(x, y) = \frac{1}{1+25(x^2+y^2)}, x, y \in (-1, 1)$. The previous study demonstrated the benefit of adaptive learning rate, especially those combining different moments[3] like ADAM. For high dependency among features, addition of regularization constrained the parameter coefficients, thus stabilizing the solution and deferring the risk of overfitting to higher complexities. When convergence speed becomes an issue, Mini-batch and Stochastic Gradient Descend should be used, but this inevitably degrades the performance. We are going to prioritize accuracy and close fit, when further analyzing the effect of regularization, learning rate, and network architecture i.e. depth, width, and types of activation functions.

A separate study will concern how ANNs can be used for classification. We consider the standard Wisconsin Breast cancer and Iris flower data sets from *sklearn* library[5][6] as binominal and multiclass problems respectively. Performance metrics will be prediction accuracy, confusion matrices, Receiver Operating Characteristic (ROC) curve and Gain curves. The earlier mentioned optimization and architecture attributes will be tested to achieve best performance.

II. THEORY AND METHOD

A. Setting up and using Neural Networks

In this section we define the general network build and the universal procedure for optimization, common

[1] The universal approximation theorem states that a neural network with a single hidden layer containing a finite number of nodes can approximate any continuous function on a compact subset of $\mathbb{R}^n$, provided the activation function is non-linear and bounded[[1].

[2] The Runge function is a case where polynomial regression of increasing complexity fails to converge, exhibiting amplified oscillations near the edges of the interval. Accurately tracing the shape near the edges, without collapsing the central fit, proved to be a major challenge[2][3].

[3] Setting the learning rate a function the gradient, makes GD methods sensible to the topology of parameter space. Here, moment refers to an exponential moving average of past gradients. Different formulations of moments are best suited for distinct topologies[2][4].

for both regression and classification usage. We start by assuming arbitrary input and therefore leave activation and cost functions ambiguous.

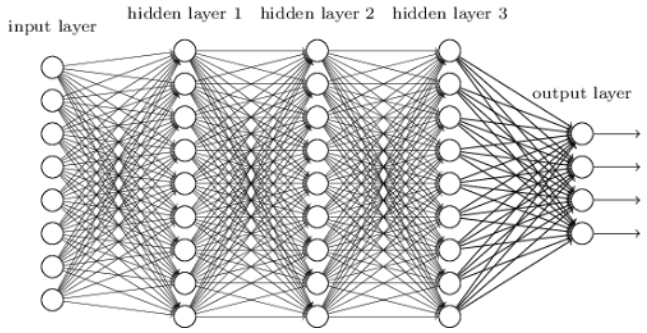


Figure 1: Neural Network with 3 hidden layers. Each node in layer l has n_{l-1} receiving and n_{l+1} transmitting connections, each with specific weight parameter W_{ij}.

A neural network is composed of L layers, such that $l = 0$ is the input layer, while $l = L$ is the final output layer. Layers in between are called "hidden", and the term "depth" refers to the number of hidden layers. Each layer contains n_l nodes which connect as shown in Fig.1. The number of nodes or inter-connections is said to be the "width" of the network. For a general ANN to comprehend both multidimensional and classification data, we define the batched input:

$$X \equiv a^0 \in \mathbb{R}^{n_0 \times m} \tag{1}$$

where m denotes the number of samples in the batch, and n_0 is the number of nodes in the input layer corresponding to data features. Note the difference from a fixed complexity P of a conventional feature matrix $X^{N\times P}$. The complexity of ANNs is not determined by input dimensions, but the entire architecture. Initially arbitrary, through optimization, the functional model assumes the form of the responsive network. On the other end, the number of output nodes corresponds to the dimensionality of the test data, and reflects the outcome for the particular problem, f.ex. number of classes for a classification problem.

Each node in layer l has n^{l-1} connections with the previous layer. When a node receives an impulse vector $a^{l-1} \in \mathbb{R}^{n_{l-1}\times 1}$, it computes the weighted sum z^l and generates a response $a^l = \sigma^l(z^l)$ where σ^l is some arbitrary layer-specific activation function. The response of the jth node in layer l is given by:

$$z_j^l = \sum_{i=1}^{n_{l-1}} W_{ij}^l a_i^{l-1} + b_j^l, \quad a_j^l = \sigma^l(z_j^l) \tag{2}$$

where W, b are trigger parameters defined as:

$$\Theta^l = \{W^l, b^l\}, \quad W^l \in \mathbb{R}^{n_l \times n_{l-1}}, \quad b^l \in \mathbb{R}^{n_l \times 1} \tag{3}$$

W_{ij}^l is the weight of the connection from node i in layer $l-1$ to node j in layer l, while b_j^l is a bias term that allows neuron response even when its weighted input is zero. This prevents a premature break in the training process. The task of optimization is to adjust these parameters so that the network output a^L best matches some target $Y \in \mathbb{R}^{n_L \times m}$. The process of propagating signals through the network is called Feed-Forward (FF). The model output is evaluated using some cost function $\mathcal{C}(a^L, Y; \Theta)$. We have so far avoided specifying activation functions, but assuming they are non-linear, we use Gradient Descend methods to minimize the cost with respect to the trigger parameters. Because of several nested dependencies, calculating gradients at some layer l requires repeated use of chain rule through all preceding layers:

$$\frac{\partial \mathcal{C}}{\partial (W,b)^l} = \frac{\partial \mathcal{C}}{\partial a^L}\frac{\partial a^L}{\partial \sigma^L}\frac{\partial \sigma^L}{\partial z^L}\frac{\partial z^L}{\partial (W,b)} \cdots \frac{\partial z^l}{\partial (W,b)^l} \tag{4}$$

This procedure is called Back Propagation. Defining $\delta_j^l \equiv \frac{\partial \mathcal{C}}{\partial z_j^l}$, we can regularize the steps. Since,

$$\frac{\partial \mathcal{C}}{\partial z_j^{l-1}} = \sum_k \frac{\partial \mathcal{C}}{\partial z_k^l}\frac{\partial z_k^l}{\partial z_j^{l-1}}, \quad \frac{\partial z_k^l}{\partial z_j^{l-1}} = W_{ij}^l \frac{\partial \sigma^{l-1}}{\partial z_j^{l-1}} \tag{5}$$

where we took the derivative of eq.2, we can express:

$$\delta_j^{(l-1)} = \sum_{k=1}^{n_l} \delta_k^l W_{kj}^l \frac{\partial \sigma^{l-1}}{\partial z_j^{l-1}} \tag{6}$$

Using $\frac{\partial z_j^l}{\partial W_{ij}^l} = a_i^{l-1}$ and $\frac{\partial z_j^l}{\partial b_i^l} = 1$, the GD layer update:

$$\begin{aligned} W_{jk}^l &\leftarrow W_{jk}^l - \eta' \frac{\partial \mathcal{C}}{\partial W_{jk}^l}, \quad \frac{\partial \mathcal{C}}{\partial W_{jk}^l} = \delta_j^l a_k^{l-1} \\ b_j^l &\leftarrow b_j^l - \eta' \frac{\partial \mathcal{C}}{\partial b_j^l}, \quad \frac{\partial \mathcal{C}}{\partial b_j^l} = \delta_j^l \end{aligned} \tag{7}$$

where η' is some adaptive learning rate computed by one of the algorithms defined in[2][4]. The $a^l, l \in [1, L]$ are known from FF, thus updating trigger parameters goes backwards $L \to 1$ and $\delta_j^L = \frac{\partial \mathcal{C}}{\partial a_j^L}\frac{\partial \sigma^L}{\partial z_j^L}$ will have unique dependency on the cost and final activation functions.

B. Defining Cost Functions

In this section we specify the different activation and cost function that will be used in this study. For regression problem with expected continuous output we keep the Mean Squared Error:

$$\mathcal{C}_{MSE} = \frac{1}{n_L}\sum_{j=1}^{n_L}\left(a_j^L - Y_j\right)^2, \quad \frac{\partial \mathcal{C}}{\partial a_j^L} = \frac{2}{n_L}(a_j^L - Y_j) \tag{8}$$

For classification, the output represents a probability vector, hence, the cost function (Cross Entropy) should depend on the negative logarithm of the probability:

$$\mathcal{C}_{CE} = -\frac{1}{n_L}\sum_{j=1}^{n_L}\sum_{c=1}^{C} Y_{j,c}\log\big(a^L_{j,c}\big), \tag{9}$$

where $Y_{j,c}$ and $a^L_{j,c}$ are probability vectors, with c referring to the class label. Binary classification is then a special case where $C = 2$. This formulation implicitly includes the complementary probabilities $(1-Y)\log(1-a)$ *only* if the final activation function is Softmax (see eq.14), where complementarity is ensured by the normalization condition. For this combination, both multiclass and binary backpropagation enjoy a useful simplification[7]:

$$\delta^L_j = \frac{\partial \mathcal{C}_{CE}}{\partial \mathcal{S}^L_j}\frac{\partial \mathcal{S}^L_j}{\partial z^L_j} = \frac{1}{n_L}(a^L_j - Y_j) \tag{10}$$

Regularization can be included in either of the previous cost functions by addition of an extra L1 and L2 norm terms with regularization hyperparameter λ:

$$\mathcal{C}^{\text{regularized}}_{\text{MSE/CE}} = \mathcal{C}_{\text{MSE/CE}} + \lambda \begin{cases} \|W\|_2^2 & \text{(L2: Ridge)} \\ \|W\|_1 & \text{(L1: Lasso)} \end{cases} \tag{11}$$

Since regularization terms depend solely on W, we need to update only stage 7, note the layer index:

$$\frac{\partial \mathcal{C}^{\text{reg.}}}{\partial W^l_{jk}} = \delta^l_j a^{l-1}_k + \lambda \begin{cases} 2W^{l-1}_{jk} & \text{(L2: Ridge)} \\ \text{sgn}(W^{l-1}_{jk}) & \text{(L1: Lasso)} \end{cases} \tag{12}$$

C. Defining Activation Functions

The universal approximation theorem requires activation functions to be, among other things, continuous and bounded[*]. Bounded range is required to comprehend probability vectors, but it can also serve as an extra level of stability by preventing outputs from blowing up. Had all activation functions been linear, the overall mapping could be expressed as *one* effective linear transformation $Y = W_{\text{eff}}X + b_{\text{eff}}$, thus, the network would not be able to solve non-linear problems. For non-linear transformations, a classical bounded choice is the Sigmoid function:

$$\sigma(z) = \frac{1}{1+\exp(-z)}, \quad \frac{\partial \sigma(z)}{\partial z} = \sigma(z)\big(1-\sigma(z)\big) \tag{13}$$

with output range $\in (0, 1)$, it fits naturally in the context of probability. The Sigmoid can be generalized to a case of C classes using the Softmax function:

$$\mathcal{S}(z^l_i) = \frac{\exp\big(z^l_i\big)}{\sum_{c=1}^{C}\exp(z^l_c)}, \quad \frac{\partial \mathcal{S}}{\partial z^l_j} = \mathcal{S}(z^l_j)\big(\delta_{ij} - \mathcal{S}(z^l_j)\big) \tag{14}$$

The normalization ensures that $\mathcal{S}(z_i)$ represents a probability vector where $\sum_{i=1}^{C}\mathcal{S}(z_i) = 1$. Therefore, it is a more universal function that can be reduced to a binary case by setting $C = 2, i = j$. Both the Sigmoid and the Softmax are "bottle neck" functions that compress the data range. Hence, they are natural for final steps, where Softmax has earlier mentioned benefits.

Both functions can cause vanishing or exploding gradient problems, resulting in extreme training tempo. To mitigate this, common techniques include Gradient Clipping: $\frac{\partial \mathcal{C}}{\partial \Theta} = \min(\frac{\partial \mathcal{C}}{\partial \Theta}, \text{cap val.})$ or placing Batch Normalization before the activation function: $z \leftarrow z - \max||z||$.

For simple transmission of signals between hidden layers, one can use linear activation $\sigma(z) = z$, or more commonly, the Rectified Linear Unit: $\text{ReLU}(z) = \max(0, z)$. Since the ReLU gradient is zero for negative z, a node can get "deactivated" by a negative weighted input and never recover. Large initial learning rates might deactivate whole parts of the network for no reason. That is why leaky ReLU is used instead: $\text{LReLU}(z) = \max(\alpha z, z)$, where α is some parameter that allows recovery, as the gradient gradually leaks into the positive domain.

D. Evaluating Classification

Given the final output a^L and labeled target Y, both in $\mathbb{R}^{m\times C}$, the classification accuracy is defined as:

$$\text{Accuracy} = \frac{1}{m}\sum_{i=1}^{m}\mathbb{I}(\hat{y}_i = y_i), \quad \hat{y}_i = \arg\max_c(a^L_{ic}) \tag{15}$$

where $\mathbb{I}$ is the indicator function that increments when the predicted class matches the target. To analyze performance per class, we use confusion matrices:

$$\text{Binary: } \begin{pmatrix} \text{TP} & \text{FP} \\ \text{FN} & \text{TN} \end{pmatrix} \quad \text{Multiclass: } \begin{pmatrix} n_{11} & n_{12} & \cdots & n_{1C} \\ n_{21} & n_{22} & \cdots & n_{2C} \\ \vdots & \vdots & \ddots & \vdots \\ n_{C1} & n_{C2} & \cdots & n_{CC} \end{pmatrix}$$

Here, rows and columns represent actual and predicted classes respectively. This way, diagonal elements n_{ii} correspond to correct classifications, and off-diagonal $n_{i\neq j}$, misclassifications. A different evaluation scheme, instead of taking the argmax, evaluates based on whether the probability for class i surpasses some confidence threshold t, i.e. $\mathbb{I}(p_i \geq t) = \text{true}$. By sweeping $t \in (0, 1)$ and plotting the True Positive Rate (TPR) against the False Positive Rate (FPR) for each threshold, produces the so called Receiver Operating Characteristic (ROC) curve.

$$\begin{aligned} \mathrm{TPR}_c(t) &= \frac{\mathrm{TP}_c(t)}{\mathrm{TP}_c(t) + \mathrm{FN}_c(t)} \\ \mathrm{FPR}_c(t) &= \frac{\mathrm{FP}_c(t)}{\mathrm{FP}_c(t) + \mathrm{TN}_c(t)} \end{aligned} \tag{16}$$

The ROC curve for multiclass classification extends the binary case by iterating over all classes, treating each class in turn as positive while aggregating all other as negative. An ideal classifier maximizes TPR and minimizes FPR, giving a curve that approaches the top-left corner. Diagonal ROC curve describes random guessing. The Area Under the Curve (AUC) quantifies this performance, ranging from 0.5 (random guessing) to 1.0 (perfect classification). Multiclass AUC is calculated by:

$$\mathrm{AUC}_{\mathrm{multiclass}} = \frac{1}{C}\sum_{c=1}^{C}\int_0^1 \mathrm{TPR}_c\big(\mathrm{FPR}_c(t)\big)dt \tag{17}$$

Finally, the cumulative gain curve shows the fraction of positive cases captured when targeting the top α percent of samples ranked by predicted probability. For example: We study N patients for cancer. We select αN "most suspicious" according to some model, and count how many test positive: $P(\alpha)$. This is compared to P cases of cancer cases among N. If the ratio is small, our model is not efficient for determining cancer, not necessarily due to poor features, but because it has low detection rate i.e lacking prioritization of rare positives.

$$\mathrm{Gain}(\alpha) = \frac{1}{C}\sum_{c=1}^{C}\frac{P_c(\alpha)}{P_c} \tag{18}$$

Here, $P_c(\alpha)$ is the proportion of true positives for class c in the top α fraction of predictions, and P_c is the total number of positives for class c. A sharp gain curve indicates a model that efficiently detects positive cases.

E. Preprocessing

Before training, the batched input must be proecessed, we standardize each feature column to have zero mean and unit variance. This is done to ensure that each feature initially has equal weighting in the analysis. This is necessary when applying regularization, to ensure even penalization across features. The data is split into training and test subsets using a standard 1:4 ratio. The Model is optimized using the training data, while the test data is used to evaluate how good the model is at generalization by predicting unseen data. Before training, we scale using a 'standard scaler' provided by scikit-learn library to scale X_{test} and X_{train}. We subtract the mean value from Y_{test}, which will be added later during evaluation as an offset. This prevents scaling from introducing bias into the optimization.

F. Resources [under construction]

For the simulations, we wanted to rely as much as possible on our own code, ensuring full control over the methods, scaling and resampling techniques. The same preprocessing scheme was used across all comparisons, so that any observed differences can be attributed solely to the parameter we changed on purpose. Library usage:

- **Sklearn-learn**: Wisconsin Breast Cancer data[5], Iris Flower data[6],
- **Scikit-learn**: StandardScaler, `train_test_split`, accuracy_score, label_binarize, RocCurveDisplay, roc_curve, auc and resample.
- **NumPy, autograd**: grad, norm, pseudoinverse
- **Matplotlib, Seaborn**: plot generation.

We also made active use of OpenAI: ChatGPT and DeepSeek to identify bugs and logical inconsistencies in both the code and our comparative approach.

The code with necessary packages can be accessed here: https://github.com/4Lexium/Data-Analysis-and-Machine-Learning/tree/main/project1

III. RESULTS AND DISCUSSION [UNDER CONSTRUCTION]

A. Runge 1D

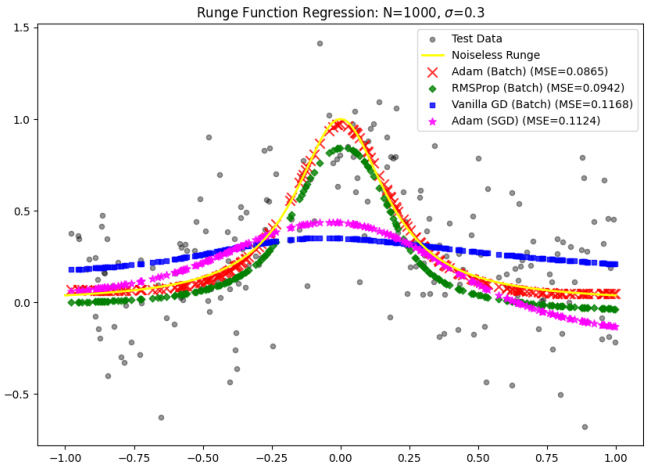


Figure 2: The plot displays a comparative performance of different optimization methods for back propagation for a fixed Network architecture and stochastic data. As predicted the ADAM adaptive method using different momentum calculation is best suited for the non-trivial shape. Stochastic Gradient Descend does severely worsen the fit around the center, but improves the convergence rate. For this study we prioritize accuracy and close fit, hence ADAM in Batch mode will be used onward.

B. Runge 2D

Big data, machine learning, artificial intelligence, digital manufacturing, big data analysis, quantum communication and internet of things see Fig.**??**.

C. Iris Data: Multiclass classification

bla bla bla flowers Fig.6. Only 4 features, very dependent.

D. Cancer Data: Binary classification

bla bla bla using log to distinguish distributions Fig.5.

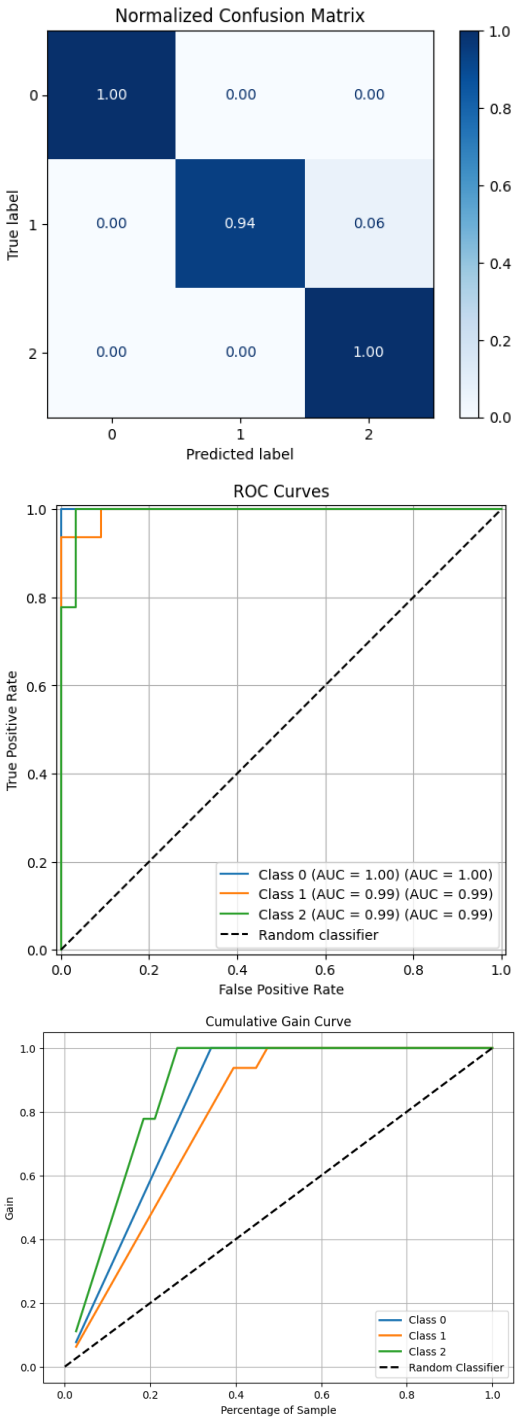


Figure 3: (Top to bottom): Classification results for Iris multiclass. Explanation of Confusion matrix, Explanation of ROC curve, Explanation of Gain curve). How do we interpret the performance based on this (how curves show more nuanced, uncover issues that are not seen in confusion or accuracy itself). Link the gain curve to the problem with features we have for flowers. They will be expressed for best performing architecture and optimization parameters. Those will be determined in an earlier plot where we study plain accuracy.

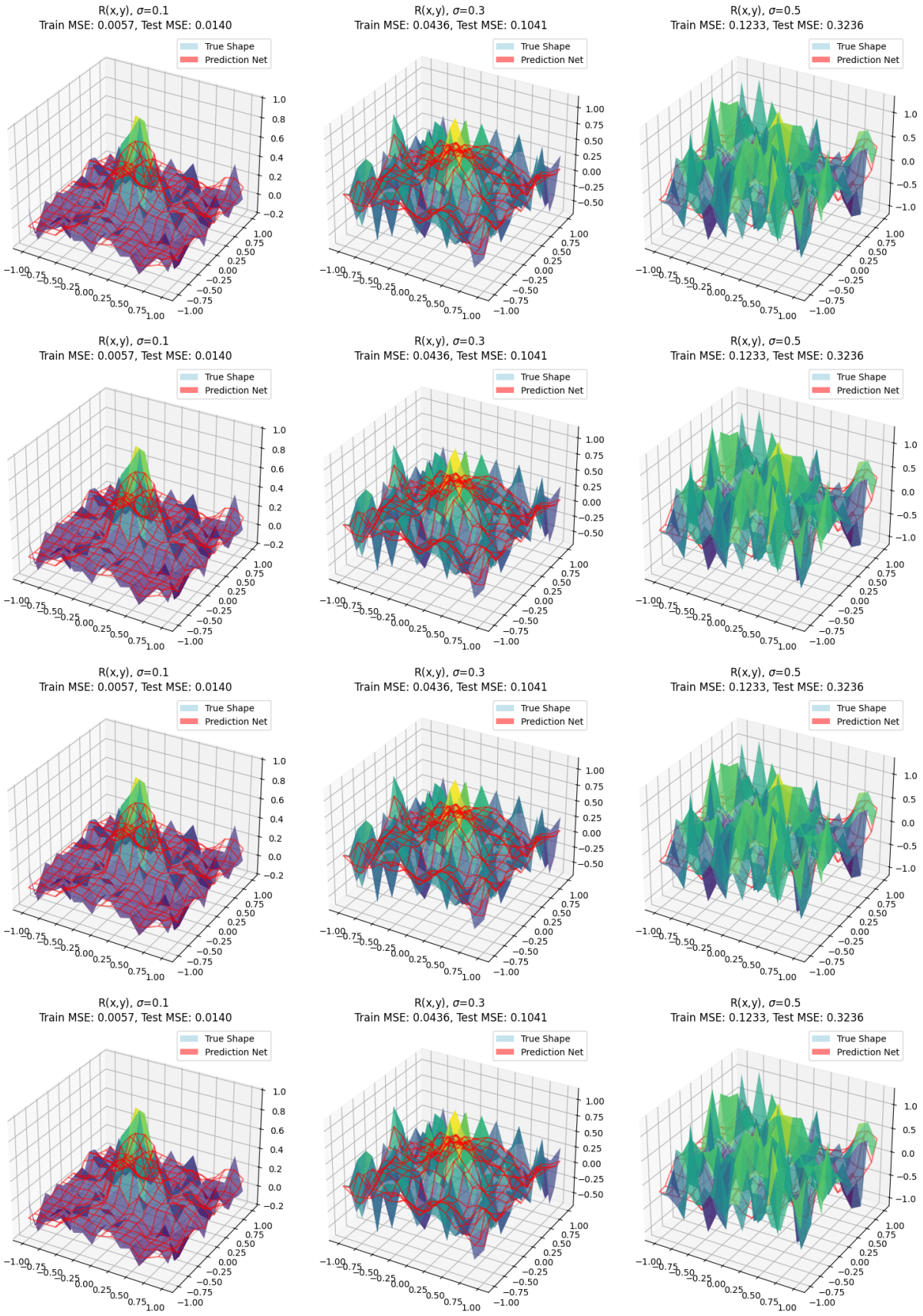
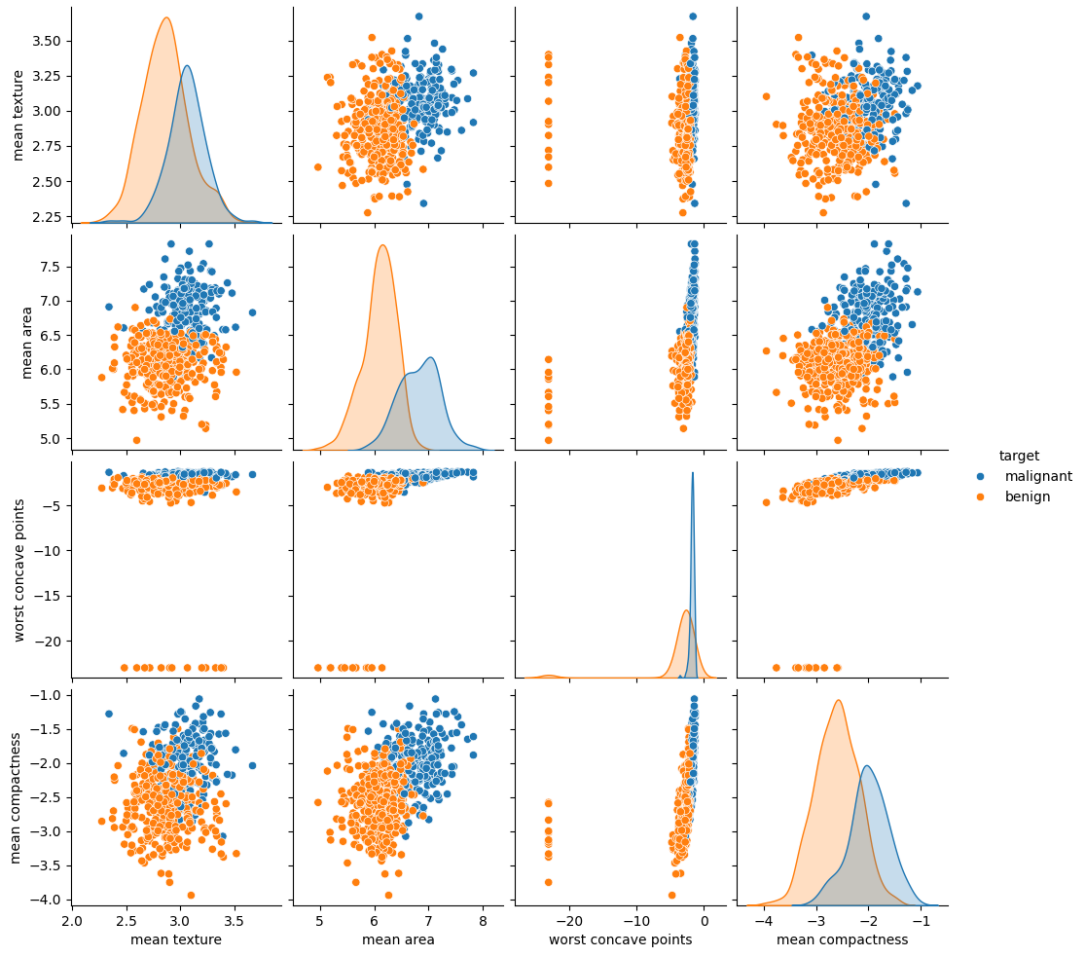


Figure 4: (From top to bottom) The grid maps show the fit of two dimensional Runge function $R(x, y)$ for different strength of the noise component. We expect ideal performance to describe the peak, and ignore the noisy peaks. An overfitted model gets distracted and learns the noise. Top graph is regular ADAM without regularization in batch mode, below regularization is introduced (result ...). The two graphs below show performance changer when the Network architecture is changed. Deeper network (more layers:... overfitting?). Wider network: (...). Regularization does (...). Comparing the train vs test MSE quantifies the degree of overfitting or how the model generalizes the true pattern of the data.



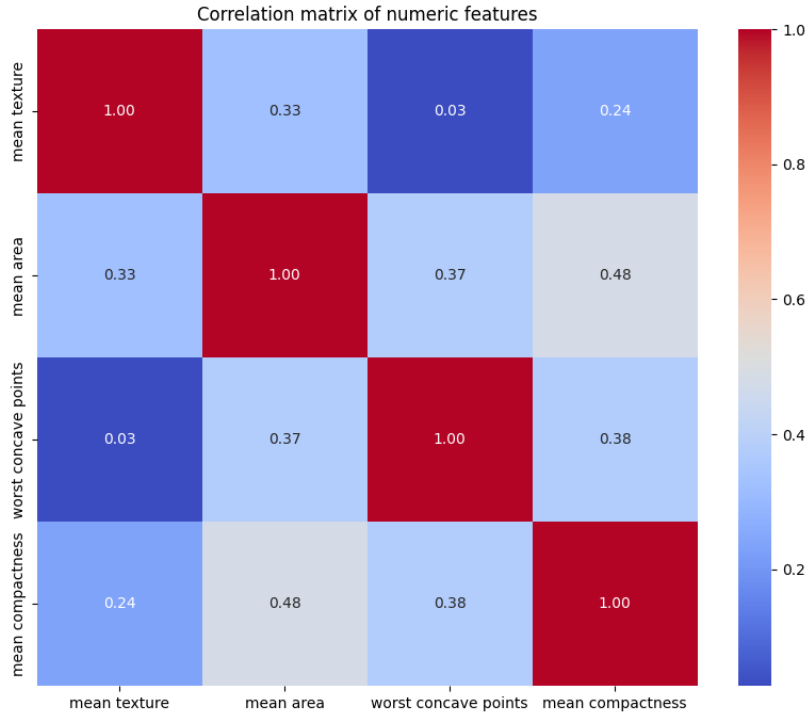


Figure 5: (From top to bottom) Describe the selection of features, we wanted to achieve as little correlation as possible, why... Why do we use log to separate the discriminating distributions. We could use more features, but we stick to 4 to make a fair comparison to flowers (they have limited features).

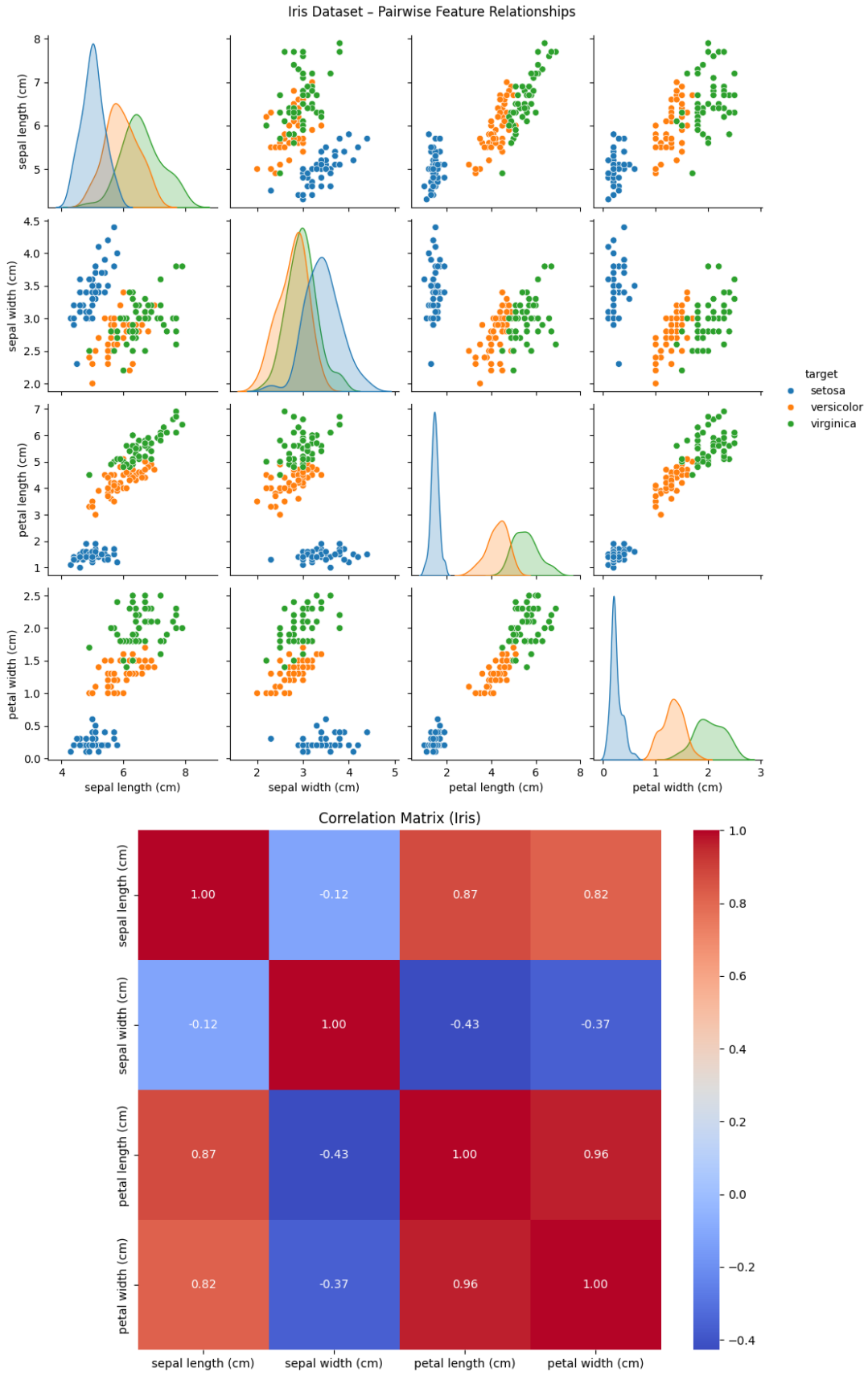


Figure 6: (From top to bottom)Iris data, describe the correlation function and of the fixed 4 parameters. Here we have to deal with high dependency among features, and only 4 features. We cant do feature selection. Why did we not use log here. Investigate use of regularization for standardized input. It should help according to our hypothesis.

IV. CONCLUSION [UNDER CONSTRUCTION]

Finally, stochastic and mini-batch GD demonstrated accelerated convergence, but at the cost of noisier updates resulting in a worse overall fit. The bell shape broadens, degrading specifically near the origin, though the edge behavior is as impressive. Notably, SDG with Importance Sapling expresses this tradeoff the most. Mini-batching approximated batch GD the most, as a faster yet more noisy variant. In general, for larger problems where convergence is essential or if only certain regions are important to fit precisely, mini-batching and SDG have advantages to consider.

[1] M. Hjorth-Jensen, *Computational Physics Lecture Notes 2015* (Department of Physics, University of Oslo, Norway, 2015), week: 37, 41, 42, 43, URL https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/intro.html.

[2] A. Umansky, FYS-STK3155 - Project 1 (2025), unpublished project report, University of Oslo, URL https://github.com/4Lexium/Data-Analysis-and-Machine-Learning/tree/main/project1.

[3] A. Peirce, *Runge phenomenon and piecewise polynomial interpolation (lecture 3)* (2017), accessed October 5, 2025, URL https://personal.math.ubc.ca/~peirce/M406_Lecture_3_Runge_Phenomenon_Piecewise_Polynomial_Interpolation.pdf.

[4] I. Goodfellow, Y. Bengio, and A. Courville, in *Deep Learning* (MIT Press, 2016), accessed October 5, 2025, URL https://www.deeplearningbook.org/contents/optimization.html.

[5] scikit-learn Developers, *sklearn.datasets.load_breast_cancer: Breast cancer wisconsin (diagnostic) dataset*, https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_breast_cancer.html (2025), accessed 2025.

[6] scikit-learn Developers, *sklearn.datasets.load_iris: Iris flower dataset*, https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_iris.html (2025), accessed 2025.

[7] T. Kurbiel, *Derivative of the softmax function and the categorical cross-entropy loss*, https://medium.com/data-science/derivative-of-the-softmax-function-and-the-categorical-c (2021), medium – TDS Archive; accessed 2025.